Arrow: A Modern Reversible Programming Language

Author:
Eli Rose

Advisor:
Bob Geitz

Abstract

Reversible programming languages are those whose programs can be run backwards as well as forwards. This condition impacts even the most basic constructs, such as `=`, `if` and `while`. I discuss Janus, the first imperative reversible programming language, and its limitations. I then introduce Arrow, a reversible language with modern features, including functions. Example programs are provided.

April 10, 2015

Introduction:

Many processes in the world have the property of *reversibility*. To start washing your hands, you turn the knob to the right, and the water starts to flow; the process can be undone, and the water turned off, by turning the knob to the left. To turn your computer on, you press the power button; this too can be undone, by again pressing the power button.

In each situation, we had a process (turning the knob, pressing the power button) and a rule that told us how to "undo" that process (turning the knob the other way, and pressing the power button again). Call the second two the *inverse*s of the first two. By a *reversible* process, I mean a process that has an inverse.

Consider two billiard balls, with certain positions and velocities such that they are about to collide. The collision is produced by moving the balls according to the laws of physics for a few seconds. Take that as our process. It turns out that we can find an inverse for this process – a set of rules to follow which will undo the collision and restore the balls to their original states[1]. Therefore, physics is reversible in this sense.

Here are two additional ways to characterize reversibility.

Local

In order for a process to be reversible, it must preserve information from step to step. We started out with certain information about the billiard balls (their positions, masses, and velocities) and in order to reverse the collision we must still have that information after the collision has occurred, albeit potentially in a different place or form.

In this case, the momentum (a function of velocity) of the ball has been transferred into whatever it collided with. In the case of both the faucet and the power button, we can imagine one bit which stores the state. Since both states can be completely represented using only this single bit, information is neither created nor destroyed. The same is true of any reversible process.

Global

Consider a process as a function from start states to final states. Then, the process is reversible if and only if this mapping is bijective.

For the faucet and the power button, the function sends off to on and on to off. With the billiards, each pair of initial positions and velocities for the two balls leads to a different pair of final positions and velocities.

[1]For Newtonian physics, these rules are simple: just substitute $-t$ for t in the original equations. This is called *time-reversal symmetry*. Modern physics obeys what is called *CPT symmetry*, meaning that one needs to invert *C*harge (particles become antiparticles) and *P*arity (everything is reflected as in a mirror) as well as *T*ime in order to run the equations backwards [1]. But both types work equally well for us – we only care that a rule exists.

Irreversible Processes:

We often think of real-world processes such as burning firewood or shattering glass as irreversible, since they increase entropy. The definition I have given above, phrased in the language of information and states, is known as *logical reversibility* [2]. A physical process that is reversible in the sense that it does not increase entropy is called *thermodynamically reversible*[2].

These two ideas are distinct, but related by an idea known as Landauer's Principle, which is that logically irreversibility implies thermodynamical irreversibility [4]. In other words, a step that goes from many states to fewer (e.g. erasing bits or merging computation paths) unavoidably results in a certain minimum amount of entropy increase. Typically, this entropy manifests in the form of heat [4]. This is a powerful motivation for the study of reversible computing since it means that, in principle, logically irreversible computations are fundamentally inefficient.

Nearly all computer programs have this problem. Consider any assignment statement, such as `x = 0`. The statement provides no way to "undo" itself – we have no way of knowing what `x` was before it was set. Locally speaking, the information represented by the value of `x` was destroyed when `x` was overwritten. Globally speaking, the function represented by `x = 0` sends all the initial states where `x == 1`, `x == 2`, ... to the same final state where `x == 0`. It is a collapsing or homogenezing process.

Reversibility of Program vs. Step

Note that it is perfectly possible to write a program which overall computes a bijection between inputs and outputs, but in which not every step is reversible. In fact, every program can be trivially modified to satisfy this condition. Simply have the program save a copy of its entire initial state at the beginning. Then it becomes clear that the initial and final states will be in bijection, since the final state of the program will effectively be a tuple `(initial_state, final_state)`.

Following Landauer and Bennett [5] [6], we will preclude these types of programs, and require that our reversible programs be composed entirely of reversible steps. One reason is because Landauer's Principle works on the level of individual steps, and so these types of "save-the-input" programs would still expend the heat for their irreversible steps. Another way of stating this is that we want "going back a step" to be *easy* – not to involve re-running the computation from the beginning.

[2] Of course all processes create entropy. A thermodynamically reversible process can be made to create an arbitrarily small amount by running it arbitrarily slowly. [3]

Goals & Definitions:

Define:

- a *reversible program* to be one in which every step is logically reversible.
- a *reversible programming language* to be one in which every program is reversible.

Work has been done on reversible functional programming languages [7] but my focus is on imperative languages. The most natural unit of "step" for a imperative programming language is the statement. Therefore, our goal is to create a language in which every statement has an inverse.

If we can invert each statement, then we can invert the whole program by executing the inverse of each statement from the bottom up. The inverse proceeds in reverse order to the original.

```
x += 3;                    x += 4;
y *= x;        ⟺          y /= x;
x -= 4;    is the inverse of   x -= 3;
```

This is just the idea that if you put your socks on, then put your shoes on, you have to take your shoes off before taking your socks off. If we consider statements as functions applied to the program's state, it's just the rule for inverting compositions of functions: $(f(g(x)))^{-1} = g^{-1}(f^{-1}(x))$.

As seen above, the inverse of statements of the form `x` $\oplus$`= y` is easy to find: we have `x += y` $\iff$ `x -= y` and `x *= y` $\iff$ `x /= y`. The goal is to have each statement's inverse be just as easy to find.

Janus

Created in 1982 by Howard Derby and Christopher Lutz, Janus (and named after the two-faced Roman god) was the first reversible programming language. Lutz sent the language specification in a letter to Dr. Rolf Landauer four years later.

Janus has since been re-implemented by the Program Inversion and Reversible Computation group at the University of Copenhagen, with various enhancements and additions. Since there is an online interpreter for this version of the language, it is easier to experiment and observe programs. This is the version of Janus I will use as reference.

Aside from the reversibility condition, Janus is a bare-bones imperative programming language. It has no:

- Datatypes. The only allowed values are integers, either 32-bit (in which case all arithmetic is implicitly mod 2^{32}) or arbitarily large. There are no strings, floats, objects, pointers, or even structs.

- Functions. Janus has only what it calls "procedures", which differ from functions in that they return nothing, have no local scope, and use pass-by-reference for all parameters. They are a wrapper around "goto" and some variable aliases. Procedures are called as a standalone statements only, not in expressions (because there is no return statement).

Its interesting features come directly from the requirement that every statement be invertible.

- Janus has no assignment statement. It uses only `+=`, `-=` and `!=` (XOR-equals). There is no `*=` or `/=`. Since all data values are integers, and the usual integer division isn't invertible, the `/=` operator is disallowed. And since `*=` is its inverse, that is disallowed as well. This is an illustration of the principle that every statement in the reversible language must be "first-class": i.e. every statement that appears in the inverse of a program must also be valid in any other context.

- Conditionals. What is the inverse of an if-statement? The ordinary if-statement is irreversible, since when you encounter it from the bottom there is no way to decide which branch to undo. Consider the following incomplete snippet of Janus code:

```
if (x % 2 == 0) then
    x += 1
fi
```

  We can see that if `x == 2` and `x == 3` both to `3`, which is irreversible. When going backwards and trying to undo this statement, absent other information we can't tell if the `3` we're looking at should become a `2` or not.

  Janus ensures reversibility by requiring the programmer to attach an extra condition after the word `fi` at the end of if-statements. The condition should evaluate to true if and only if the if-statement's condition evaluated to true. The intention is to provide a way to tell which branch was executed on the way down; the onus is on the programmer to ensure correctness.

  Note that there is now no way to write the function above; there is no condition which holds if and only if the main branch of the if-statement was taken. We could, however, invent a way to hold onto this information.

```
if (x % 2 == 0) then                    if (even == 1) then
    x += 1                                  even -= 1
    even += 1          ⟺ is the inverse of  x -= 1
fi even == 1                            fi x % 2 == 0
```

If `even` was always 0 before reaching this code, then our if-statement would become invertible: the `even == 1` condition tells us which branch to take. This doesn't mean that we need to store the outcome of every if-statement. In another context, we might not need to invent a variable to store this information for us – often one already exists. Examples of these types of if-statement conditions will appear later in the example programs.

- Loops. What is the inverse of a while-loop? Intuitively, we want to undo the statements inside the while-loop as many times as we did them forwards. As in if-statements, Janus requires another condition. Firstly, instead of `while <condition> <statements>`, Janus uses `<statements> until <condition>`. It then requires a start condition at the start of the loop. This condition must be true when and only when the loop starts, just as the `until` condition is true when and only when the loop stops.

  These are called *from*-loops, since they start from one condition and go to another. Here an example which calculates (and uncalculates) the sum of the numbers 1 through 9.

```
from (i == 0) loop                    from (i == 10) do
    sum += i                                i -= 1
    i += 1          ⟺ is the inverse of     sum -= i
until (i == 10)                       until (i == 0)
```

- Local variables. All variables in original Janus are global, but in the University of Copenhagen interpreter you can allocate local variables with the `local` statement. The inverse of the `local` statement is the `delocal` statement, which performs *deallocation*. When inverted, the deallocation becomes the allocation and vice versa. In order to invert deallocation, the value of the variable at deallocation time must be known, so the syntax is `delocal <variable> = <value>`. Again the onus is on the programmer to ensure that the equality actually holds. Here is an example that uses a temporary variable to double an integer (since Janus has no `*=` operator).

```
local z = a                           local z = a / 2
a += z                  ⟺ is the inverse of   a -= z
delocal z = a / 2                     delocal z = a
```

- Uncalling procedures. Since computing inverses is so easy it can even be done at runtime, Janus potentially has an advantage over irreversible languages even from a purely *programmer's* perspective. Procedures are called with the `call` statement, whose inverse is the `uncall` statement, and since every statement is first-class, the `uncall` statement can be used in ordinary code as well. To uncall a procedure is to undo all of the

statements in that procedure, starting at the bottom and proceeding to the top.

There are many potential uses, such as recursive backtracking and recovering from errors. Some of the example Arrow programs later will uncall functions.

Limitations of Janus:

Janus is a fascinating proof-of-concept language; however, it suffers from severe limitations already noted above, such as a lack of datatypes and functions. As a result, Janus programs tend to be long, clunky, low-level and difficult to read.

Here is an example Janus program which compresses an array using run-length encoding.

```
procedure encode(int text[], int arc[])
    local int i = 0
    local int j = 0
    from i = 0 && j = 0 loop
        arc[j] += text[i]
        text[i] -= arc[j]
        from arc[j+1] = 0 do
            arc[j+1] += 1
            i += 1
        loop
            text[i] -= arc[j]
        until arc[j] != text[i]
        j += 2
    until text[i] = 0

    // i & j should be cleared
    from arc[j] = 0 do
        j -= 2
        i -= arc[j+1]
    until i = 0
    delocal int j = 0
    delocal int i = 0

procedure main()
    int text[7]
    int arc[14]

    text[0] += 1
    text[1] += 1
    text[2] += 2
    text[3] += 2
    text[4] += 2
```

```
text[5] += 1
call encode(text, arc)
```

It sends the array `text == [1, 1, 2, 2, 2, 1]` to the array `arc == [1, 2, 2, 3, 1, 1]` – where each digit is followed by the number of times that digit appears. (The `do` statement means to execute the following once before starting the loop, as opposed to the code following `loop` which is the body of the loop).

The aim of Arrow is to extend Janus in a way that makes it easier to read and write. This involves solving theoretical problems (what is the inverse of a return statement?) and design problems (what abstractions would help to express this common idea?).

Arrow:

Arrow, like Janus, is a reversible, imperative, interpreted language.

Datatypes: Unlike Janus, Arrow uses several datatypes: Num, List, String and Boolean.

Nums are infinite-precision rational numbers, introduced in order to allow `*=` and `/=` operators by making the operations of division and multiplication truly each others' inverses (possible loss of precision makes floating-point operations irreversible).

Lists are dynamically resizing arrays, like Java's `ArrayList` or Python's `List` (which they are based on). Lists may contain data of any type, or of mixed types. Like Python lists, they also function as stacks, with `push` and `pop` being each other's inverses.

Strings and booleans are standard. Strings are immutable.

Syntax: Syntax is inspired by Janus, but is often designed to resemble more modern languages. A statement can be `+=`, `-=`, `*=`, `/=`, which all have the obvious inverses, a `result` or `enter` or `exit` statement, which will be discussed in the next section, or any of the following:

`<id> := <expr>`	$\overset{\text{is the inverse of}}{\Longleftrightarrow}$	`<id> == <expr>`
Creates the new variable `<id>` and initializes it to the value of `<expr>`. This is the equivalent of Janus's `local` statement.		Deallocates the variable `<id>` if it is equal to `<expr>`. This is the equivalent of Janus's `delocal` statement.

```
from <exprA> {
  <statements>
} until <exprB>
```

$\Longleftrightarrow$ is the inverse of

```
 from <exprB> {
un(<statements>)
 } until <exprA>
```

Same as Janus's `from` loop.

```
for <id> := <exprA> {
    <statements>
} <inc>, until <id> == <exprB>
```

$\Longleftrightarrow$ is the inverse of

```
for <id> := <exprB>, un(<inc>) {
    un(<statements>)
} until <id> == <exprA>
```

A `for` loop is a `from` loop where the start condition allocates a variable and the end condition deallocates one. `<inc>` is usually a mod-op like `i += 1`.

Note that the increment statement has switched positions. Either way is valid, and they have different meanings.

If `<inc>` is above the body, it gets executed before the body; if it is below, it gets executed after the body and before the test. The difference is just like that between `while` and `do-while` loops.

By way of example, here is a for-loop that iterates through a list `A` and operates on its elements.

```
for i := 0 {
    A[i] *= 2
} i += 1, until i == A.length()
```

Note that this loop's inverse is the following:

```
for i := A.length(), i -= 1 {
    A[i] /= 2
} until i == 0
```

If we didn't move the increment statement, we would have the following code:

```
for i := A.length() {
    A[i] /= 2
} i -= 1, until i == 0
```

which differs in meaning (in particular, it tries to access `A[A.length()]`, resulting in an error.)

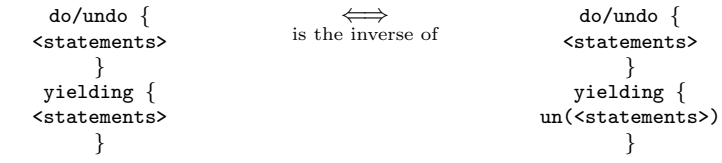
```
if <exprA> {
    <statements>
} => <exprB>
else {
    <statements>
}
```

$\overset{\text{is the inverse of}}{\Longleftrightarrow}$

```
if <exprB> {
    <statements>
} => <exprA>
else {
    <statements>
}
```

Analogous to Janus's if-statement. The rocket symbol `=>` is used before the `fi`-condition. The else is optional.

Because the case where `<exprA> == <exprB>` comes up often (whenever the code inside the if-statement doesn't affect the expression), Arrow allows `<=>` in place of `=> <exprB>`, so you don't have to write the same expression twice. Then the if-statement's inverse is itself.



For two blocks of code `A`, `B` in `do/undo {A} yielding {B}`, `do/undo` executes `A`, executes `B`, then executes `un(A)`.

Only the inner `B` block needs to be inverted, since the outer `A` blocks switch and turn into one another when inverted.

`do/undo` allows programmatic access to the inverses of statements that aren't wrapped in functions. The keyword "yielding" is used to suggest a pattern that occurrs frequently in reversible computing: do some computation to get an answer, then save the answer and undo the computation to get rid of the mess (local variables or destructive mutations) we just made. For example, in order to determine whether a number `n` is prime, you might write:

```
i := 1
from i := 1 {
    i += 1
} until n % i == 0
```

and, after this code executes, `n` is prime iff `i != n`. Now you can do something with that information, such as copy it elsewhere or return it. But you obviously don't know what `i` is at program-writing time, so how are you supposed to deallocate it?

The answer is to run the from-loop backwards to bring `i` back down to 1. Note that then deallocation is automatic. So we use `do/undo`[3].

```
do/undo {
    i := 1
    from i := 1 {
        i += 1
    } until n % i == 0
}
yielding {
    if i != n {
        // n is prime
    } <=>
}
```

[3]This could be written manually, but as the loop gets more complicated, it'll get harder for the programmer to manually determine the inverse. Wrapping the loop in a function and calling it is clunky and doesn't work with local variables.

Variables: Arrow features a basic division among variables, into **local** and **main** variables.

Main Variables: An Arrow program consists of any number of functions, one of which must be named `main`. When the program is run forwards, main is called; when it is run backwards, main is uncalled.

The main function must have one or more arguments. They look like this:

```
main(n := 3, x := [], greeting := "hello")
{
    ...
}
```

These may be though of as "arguments to main", but they differ in meaning from `argv` in C. These are the main variables. By placing them in the header for main we indicate that we are interested in the values of these variables at the end of the program. When you run an Arrow program, it prints out the values of the main variables before execution and their values after. The program can be seen as a transformation applied to these variables – the starting configuration of main variables defines a "state" which will be mapped bijectively to another "state".

Local Variables: These are allocated and deallocated during the execution of the program. No local variables exist at the beginning of the program and none remain at the end of the program.

Functions: Janus procedures are exclusively pass-by-reference and have no return statement. We consider the problems posed by each independently.

Pass-by-reference in a reversible context means that all of the arguments to a procedure call `f( ... )` must be variable names, and variable names only. A procedure's argument can't include a literal like `f(a, 1)`, or even an expression as in `f(a, b * 2)`.

A "traditional" pass-by-value function in an imperative languages works by creating local variables to hold the values of its arguments. But consider what would happen if we tried to simply drop those functions into our reversible language:

```
f(A, n){
    n += A[0]
    A[0] -= n
}

main(list := [1, 2, 3]) {
    f(A, 0)
}
```

This program will set the first item of `list` to 0 and then fail to reverse, since the 1 is not being stored in a main variable. Another way to see this is to

observe that this program will take the lists `[2, 2, 3]` and `[3, 2, 3]` to the same final state – `[0, 2, 3]`[4]. The problem is that `n` is a local variable that we don't deallocate, allowing us to leak data by transferring it to `n`.

But how could we deallocate `n`? That would mean we would have to know its value by the end of the function. Presumably that value depends on what it started out as, and the function can't see that. If we wrote `n == 1` at the end, then the function would become specialized to these particular arguments, which defeats the whole purpose of having a function.

So functions that create local variables are not reversible. But we would still like to be able to use literals or expressions as function arguments: otherwise, we'd have to create a variable, pass it in, and deallocate it just to do things like `rot_encode(data, 128)` or `process_payment(account, 10.00)`.

Arrow's solution is to divide parameters and arguments into two types: **ref** and **const**. A function header will specify one of the two for each, like this:

```
scoot(ref array, const i, ref trace) { ... }
```

ref means that a parameter is pass-by-reference, functioning identically to a parameter for a Janus procedure. **const** means that the parameter is a constant value supplied to the function. **const** is like pass-by-value except you can't modify const arguments inside the function – they're constant. No local variable is actually created, meaning it doesn't have to be deallocated.

Functions are then called with an ampersand in front of the arguments which are to be passed by reference. The ampersand is used as an analogy to C's reference-of operator. However, Arrow's `&` is not an operator as such: it's just a marker that has no meaning outside of function calls. It can only be applied to single variables, not expressions: the line `scoot(&A, &i + 1, &T)` would be a syntax error. Our function would be called like:

```
scoot(&A, 2, &T)
```

or like:

```
scoot(&A, i, &T)
```

The return statement comprises two separate parts – stopping the flow of execution, and setting a value for the function call to give back. The second is no problem: Arrow's `result` statement only sets up the return value, without stopping execution. Its inverse is itself.

`result <expr>` $\underset{\text{is the inverse of}}{\Longleftrightarrow}$ `result <expr>`

[4]There's nothing special about the list here, or even the use of a list in general. Anything mutable would have served.

Stopping the flow of execution is more problematic. Let's give the name "exit" to the statement that does this (without giving back a value). What is the inverse of `exit`? It's not a self-inverse, since executing `A` and then exiting is not undone by exiting and then executing `un(A)` – we'd stop before we undid anything. Instead, the inverse of an exit statement that tells you where to exit the function should be an "enter" statement that tells you where to enter the function. When we start executing the function, instead of starting at the top we should start at the `enter` statement.

This is what Arrow does. Since multiple `exit` statements will lead to multiple enter statements, Arrow requires a condition be attached to each `exit` statement. The exit only occurs if the condition is true. Furthermore, the conditions are required to be disjoint. Then, when the function is inverted and they become conditions on the `enter` statements, only one condition will ever be true, and we can start execution from that point.

```
exit if <expr>
```
$\overset{\text{is the inverse of}}{\Longleftrightarrow}$
```
enter if <expr>
```

In order to illustrate a subtlety of this definition, let's consider the following function and its inverse.

```
f(ref x) {
    exit if x >= 0
    x *= -1
}
```
$\overset{\text{is the inverse of}}{\Longleftrightarrow}$
```
f(ref x) {
    x /= -1
    enter if x > 0
}
```

This is in fact `x = |x|` in disguise. Going forwards, it will flip the sign only if `x` is negative. Going backwards, it will also only flip the sign if `x` is negative. We've created something irreversible; where did we go wrong?

There is an implicit `exit` at the bottom of every function. When we have no other `exit` statements, we can imagine it as `exit if True` without penalty, but, since all the conditions must be disjoint, this can no longer be true as soon as we have just one other `exit` statement.

Regardless, it still seems trivial to fill in the last condition. If there are $n-1$ other conditions, we can just OR them all together and negate the result. This will be the one "missing" condition; in this example the implicit last statement is `exit if x < 0`.

But now observe that this condition is never true at the point when the statement is encountered! By that point, `x` was either positive to begin with or it was negative and we just flipped it; either way, it can't be negative. So the implicit `exit` condition has *failed*, and it's necessary for Arrow to throw an error – in this case, a `FailureToReturnException`, unthinkable in any other language. So this function is impossible to run, and reversiblity is preserved.

Methods: Some datatypes have methods – these act like Arrow functions (they must have inverses) but are implemented at a lower level, in Python. Since Python functions can't be inverted, the inverses of methods must be hardcoded.

- Nums have the methods `is_int()` and `to_str()`, both of which are their own inverses.
- Lists have the methods `push()`, `pop()`, `peek()`, `empty()`, and `len()` (returns the length). `push()` and `pop()` are each other's inverses, and every other method is its own inverse.
- Strings have some new methods which are the inverses of operations we often do on strings. For example, if `a += b` concatenates `b` onto the end of `a`, then there must be an inverse operator `a -= b` which removes `b` from the end of `a`. Since the use of this operator constitutes a kind of assertion that `b` actually *is* on the end of `a`, Arrow throws an error if this is not the case.

  Strings have the methods `len()`, `get(index)` (returns the character at that index, as a string), `to_int()`, `+=` and `-=`. These are all their own inverses except for `+=` and `-=`. In addition, to cover the case `a = b + a`, strings have a `left_add(other)` and a `left_del(other)` method, which append/remove the other string on the left. They are each other's inverses.
- Booleans have no methods.

Future Work:

While Arrow is an improvement over Janus, it is still much more awkward and less expressive than a "real" modern language like Java or Python. This can be seen most readily by just starting to write a program in Arrow: often, one thinks something will take five minutes until requirements like the second `if`-condition emerge and necessitate spending hours instead. Even in the attached example programs, which I chose to show off the language in various ways, there are awkward work-arounds and boilerplate code.

I suspect that a language in which reversible computations *can* be expressed just as naturally as irreversible ones exists, though it is probably even stranger to our native (irreversible) way of thinking than Arrow. I think it will take a lot more work to find; however, there are many areas in which I felt that Arrow could clearly have been improved. Some improvements are more thought-out than others.

On the design side:

- The pattern `stack.push(i); i == stack.peek()` for moving the local variable `i` to the top of a stack (or string) occurs frequently. In general, with reversibility a concept of "moving" data form one location to another is needed, for example to deallocate local variables. Something like `i => stack` could be used; however, how do we know to use the `push` method

here? What would be a clean syntax? Could it work between two variables as well, even if one isn't going to be deallocated? What if you want to apply a transformation on the data between points A and B (such as turning it into a string?).

- Can Arrow support full OOP features? Some questions that would have to be addressed: what is the inverse of creating an object (e.g. what is the inverse of `malloc`? Is it `free`?)
- I/O. It's clear that the inverse of `x = input()` is clearing the variable `x`. What does this mean if `input()` is called without assigning to a variable, like if it's used as a `const` argument to a function? It seems like `input` almost needs to create a variable. Should variables which are created be condisered "main" variables, and displayed at the end of the run?

On the theoretical side, our syntax for `exit` and `enter` has an unexpected consequence: it disallows traditional recursive functions. Consider the following Arrow function, intended to compute the factorial of x and deposit it in `acc`, which starts out with the value `1`:

```
factorial(ref x, ref acc) {
    exit if x == 0

    acc *= x
    x -= 1

    factorial(&ref, &acc)
    exit if ???
}
```

We need to provide an exit condition after the recursive call. Consider our options: the impulse is to write `x != 0`, but this actually is never true at that point in the code: by the time we're removing frames from the stack, the answer has already been computed and `x` is `0`. Our condition should be `x == 0`, but conditions must be disjoint, otherwise we wouldn't know where to enter. So this function cannot be returned from.

A possible solution comes from an unlikely source: tail-call optimization. If we tail-recurse into `factorial` instead of calling it, we could eliminate the need to come back to that stack frame and hence the need for an exit condition. So we might have an `exit into <function> if <cond>` statement (inverse: `enter from <function> if <cond>`) which 1) can only appear at the end of the function and 2) exits and calls `<function>` simultaneously.

Then the following recursive factorial function works:

```
factorial(ref x, ref acc) {
    enter from factorial(&ref, &acc) if acc != 1
    enter if acc == 1

    acc *= x
    x -= 1

    exit if x == 0
    exit into factorial(&ref, &acc) if x != 0
}
```

Note that the dual-condition structure `acc == 1, x != 0` $\iff$ `acc != 1, x == 0` was necessary. Many questions remain: can all recursive functions be rewritten this way? What if `enter from` is used without a corresponding `exit into`? How to specify constant arguments in `enter from`? (I've evaded the problem with this function, since it only has reference arguments). If an `exit into` lands at an `enter from`, do their conditions have to match? It seems like this must be the case, so what happens if they don't – runtime error, or do we just not exit? Now we're creating two "classes" of `enter`/`exit` statements – can they interact with each other, or is it only allowed to enter at an `enter from` if you left via an `exit into` and vice versa?

It seems that the most important territory to explore is just in writing programs. Almost every program out of the 19 I wrote pushed the theory in a significant direction.

Arrow Example Programs:

This program calculates the prime factors of `n` and stores them in a list. In reverse, it multiplies together all the prime factors to arrive back at `n`.

```
prime_factors(ref n, ref output){
    from output.empty() {
        i := 1

        from (i == 1){
            i += 1
        } until n % i == 0

        n /= i
        output.push(i)
        i == output.peek()
    } until n == 1
}

main(
n := 10012,
output := []
){
    prime_factors(&n, &output)
}
```

This program implements run-length encoding, like the Janus program given earlier, except the Arrow program runs on strings. In reverse, it decompresses a run-length-encoded string back to full length.

```
chunk_encode(ref chunk){
    i := 0
    char := chunk.get(0)

    from i == 0 {
        chunk.left_del(char)
        i += 1
    } until chunk.get(0) != char

    chunk += i.to_str()

    last_char := chunk.get(chunk.len()-1)
    i == last_char.to_int()
    last_char == chunk.get(chunk.len()-1)

    chunk += char
    char == chunk.get(chunk.len()-1)
}

main(data := "aaaaaabbbbccaaaaaabbaaaaaabbaaaaaaaacabbb"){
    data += "$"

    from data.get(data.len()-1) == "$" {
        chunk_encode(&data)
    } until data.get(0) == "$"

    data.left_del("$")
}
```

This program bubblesorts a short list. Since sort is not reversible (it takes many states to one) some additional output is needed – here it is in the form of a "trace" that tracks the swaps that were done while sorting. They appear in the list `trace` in the format `[-1, A1, B1, A2, B2 ...]` where the `ith` swap exchanges elements `Ai` and `Bi`. To unsort, the program simply undoes the swaps.

```
pass(ref array, ref trace){
    for i := 0 {
        if (array[i] > array[i+1]){
            array[i] <=> array[i+1]

            trace.push(i)
            trace.push(i + 1)

        } => trace.peek() == i + 1
    } i += 1, until i == array.len() - 1
}

check(const array){
    do/undo {
        counter := 0
        for i := 0 {
            if array[i] < array[i+1] {
                counter += 1
            } <=>
        } i += 1, until i == array.len() - 1
    }
    yielding {
        result (counter == array.len() - 1)
    }
}

sort(ref array, ref trace){
    from trace.peek() == -1 {
        pass(&array, &trace)
    } until check(array)
}

main(
A := [3, 8, 7, 3, 2],
trace := [-1]
){
    sort(&A, &trace)
}
```

References

[1] Greaves, Hilary, and Teruji Thomas. "On the CPT Theorem." Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics 45 (2014): 46-65. Web.

[2] Muehlhauser, Luke. "Mike Frank on Reversible Computing - Machine Intelligence Research Institute." MIRI. Machine Intelligence Research Institute, 31 Jan. 2014. Web. 07 Mar. 2015.

[3] Frank, Michael. "The Reversible and Quantum Computing Group (Revcomp)." The Reversible and Quantum Computing Group (Revcomp). University of Florida, 2003. Web. 07 Mar. 2015.

[4] Bennett, Charles H. "Notes on Landauer's Principle, Reversible Computation, and Maxwell's Demon." Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics 34.3 (2003): 501-10. Web.

[5] Landauer, R. "Irreversibility and Heat Generation in the Computing Process." IBM Journal of Research and Development 5.3 (1961): 183-91. Web.

[6] Bennett, C. H. "Logical Reversibility of Computation." IBM Journal of Research and Development 17.6 (1973): 525-32. Web.

[7] Yokoyama, Tetsuo, Holger Axelson, and Robert Gluck. "Towards a Reversible Functional Language." Reversible Computation Third International Workshop, RC 2011, Gent, Belgium, July 4-5, 2011: Revised Papers. By Alexis De Vos and Robert Wille. Berlin: SpringerLink, 2012. 14-29. Print.